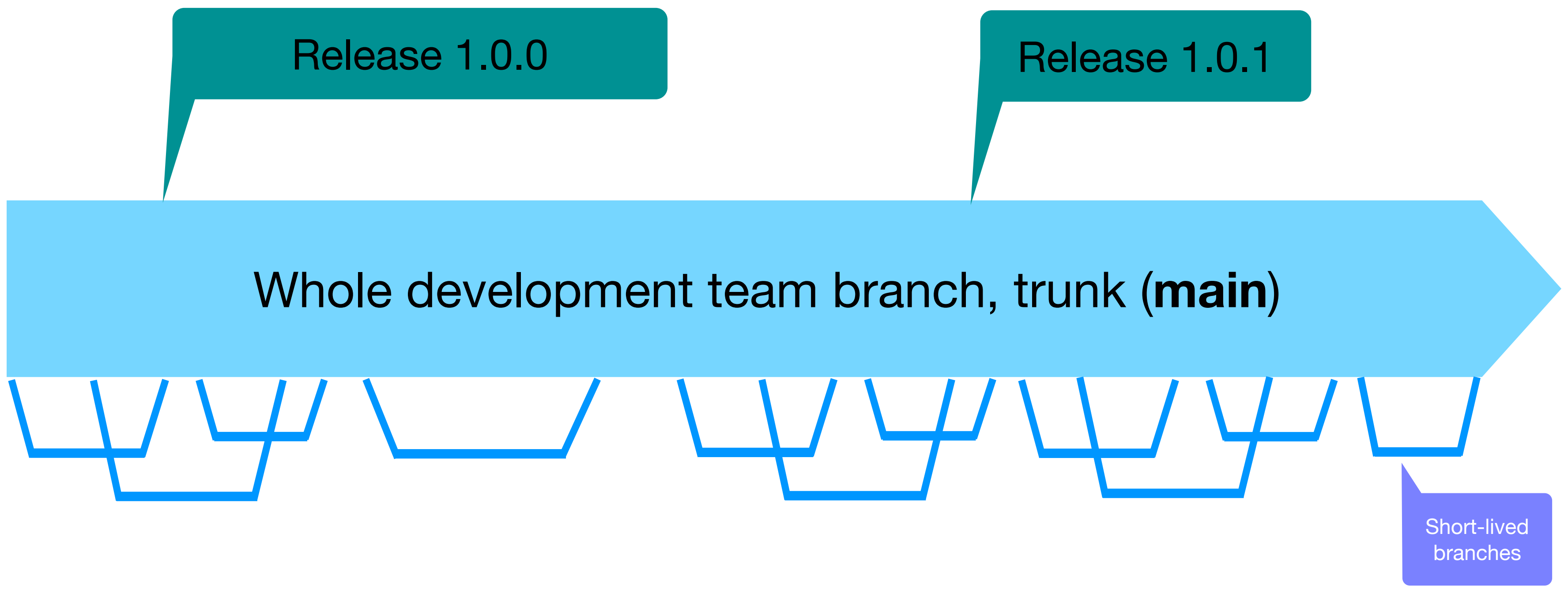
Applied CICD Patterns

The purpose of this document is to provide an strategy for **source code integration** based on Trunk Based Development and on top of that to define a **continuous delivery pattern**, from **development environment** to **production releases**. Developers can be considered as anyone contributing code, including Developers, SRE/DevOps, Testers, Architects, etc.

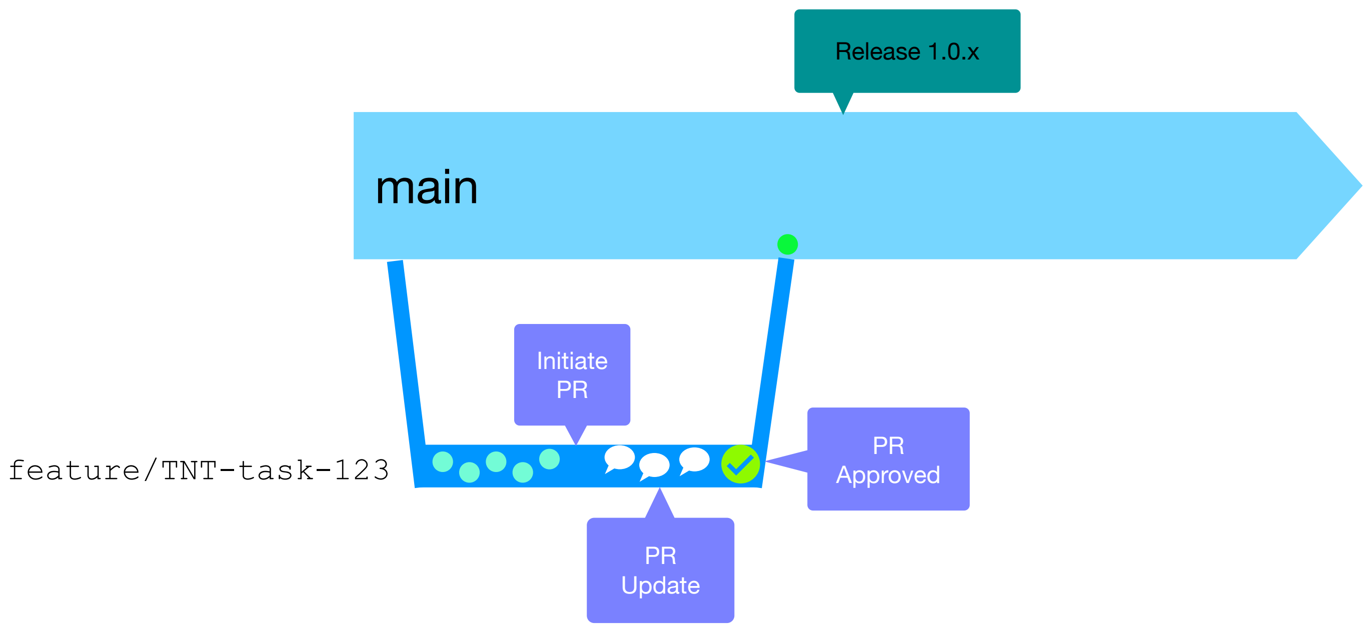
Continuous Integration

Trunk Based Development is a source-control branching model, where developers collaborate with code into a single branch called **trunk** (**master** or **main**, based on `Git` nomenclature). Developers create very short lived feature-branches that gets merged regularly into the trunk through **pull requests** that includes a peer review (**PR**), then by using a **development pipeline automation** to ensure at least, that the merge will not break the build, that the code is secure and of good quality.



Trunk Based Development at scale is best done with **short-lived feature or bugfix branches**: one person over a couple of days (max) and **flowing through Pull-Request** style code review and **build automation** before integrating (merging) into the trunk

Example of a **short-lived** feature branch and PR merge.



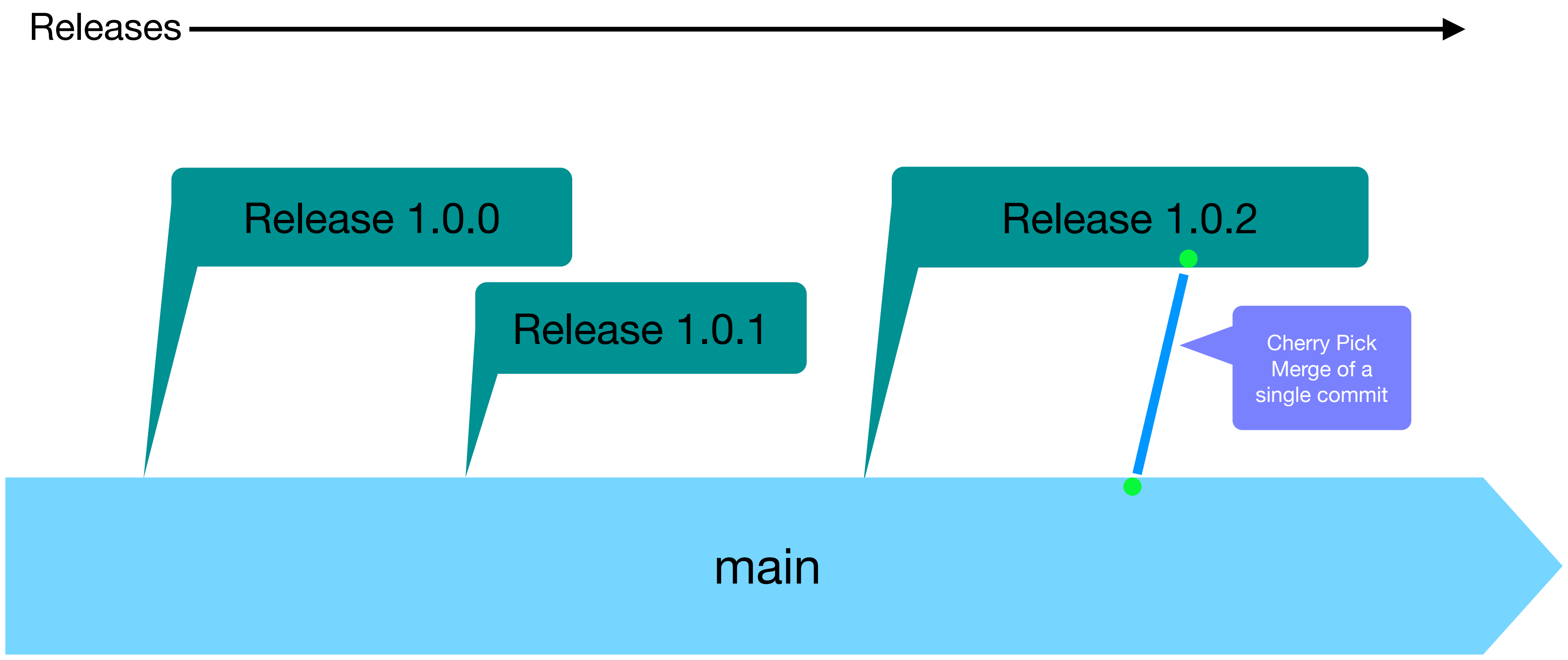
In the picture above, the branch `feature/TNT-task-123` is opened from **main**, then commits are added into it in a regular basis, then a PR is initiated, usually some comments and reviews will happen that will imply pushing new code until the approval is done, usually for at least one peer of the team, then the merge is done into **main** and the feature branch is deleted from the code base.

Proposed branch **naming** and source integration **flow**:

- **main**: For the Trunk branch, commits are not done directly to the main, its **always** via a pull-request and peer-review process.
- **feature|bugfix/<project_key>-<task_number>**: For any feature or bugfix that is new or to update as part of the related task, this imply updating pure *app code*, *tests*, *documentation*, *infrastructure as code*, *pipeline definition*, etc.
  - **<project_key>**: Correspond to a *key* or *abbreviation* for the project usually in capital case.
  - **<task_number>**: Correspond with the minimal task unit on an Agile board that is considered a *1:1 relation* with the *feature* or *bugfix* that is going to be incorporated as part of the development work. These **tickets** could have the following order of precedence:
    - **Epic > Feature > User Story > (Task, Bug)**

    - **Feature:** This ticket might have several User Stories
    - **User Story:** It could have several *tasks* and *bugfix* related, and in this case the branches will be created based on the task or bugfix number, **i.e.** for the **TNT** project, **feature/TNT-task-123 bugfix/TNT-bug-456**
- **release-<version>:** Is the release branch that is created from the **main** trunk, following the semantic versioning as candidate for **production release**, this could be the entire set of tasks of an sprint, usually 2 weeks of work, or a set of those tasks agreed during the planning that could be used for a release, this way the pattern allow the following **release cycle**:
  - **Full Sprint Release:** Making a release candidate the entire set of feature|bugfix added on the sprint.
  - **Partial Sprint Release:** A set of feature or bugfix that can be branched out as release branch for release candidate. This can also be for single commit release candidates, such practice allows faster **production** deployments during a sprint.

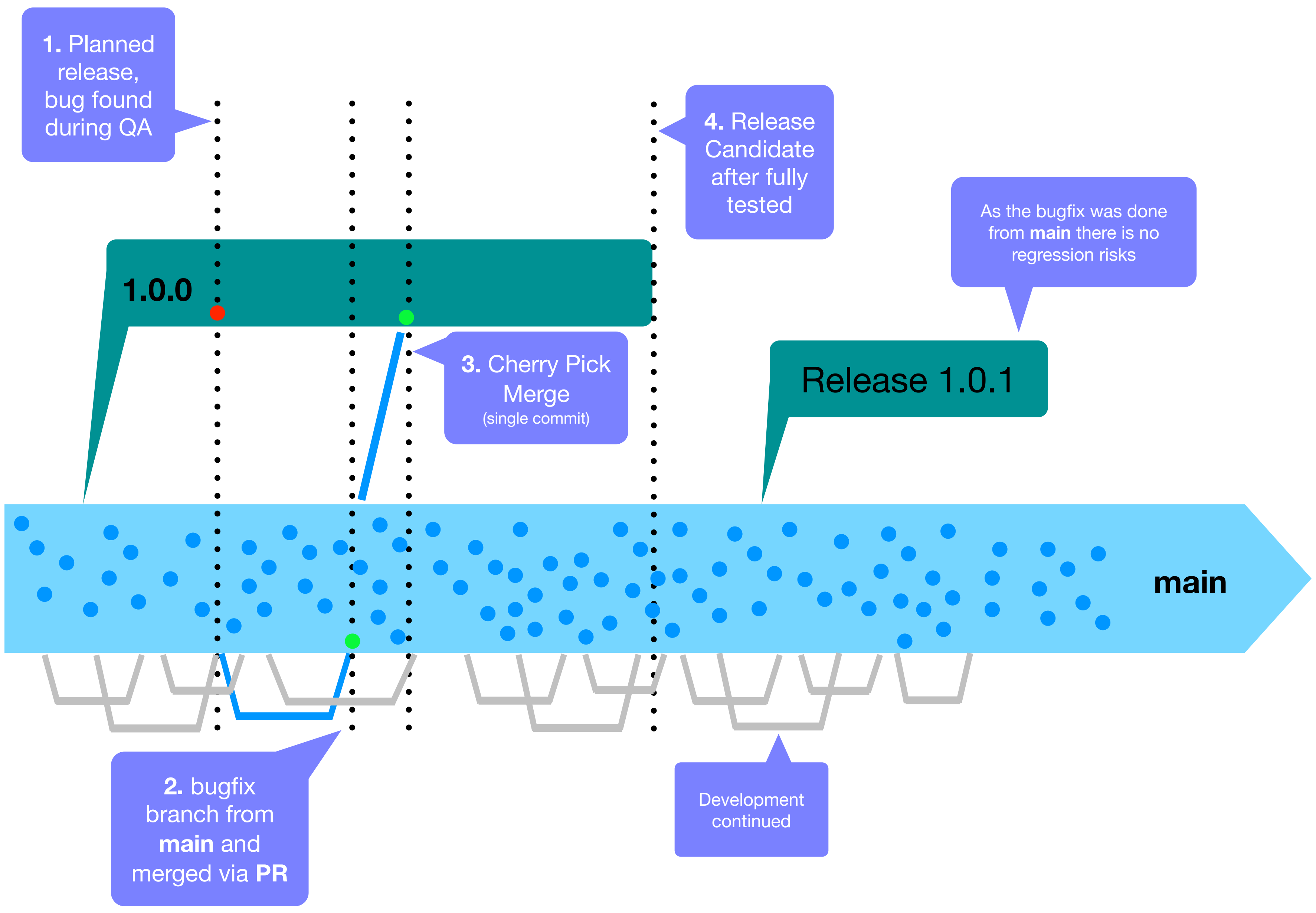
In such release branch, whether if its the full sprint, a set of features or a single feature, testing processes (QA, UAT, etc.) For **automated** and **non-automated** needs to happen on this branch, to certify a release candidate prior to the **production** deployment.



As shown in the picture, in the event that a *bugfix* has been done for an specific release branch, then such commit needs to be added from **main** into the release branch via [Git Cherry Pick](), bellow is pretty much what a bugfix flow will be:

1. **branch out** from current **main** that will be at the same point in time or more since the release branch was initially created, usually the state of **main** when the bugfix branch is created will have more context into it, as the *development team* continued since the release branch have been initially tested.
2. Once the development of the bugfix has **finished**, a peer review will be opened to get it merged into **main**.
3. Once merged, a **cherry pick** process will add such commit into the specific release branch.
   Note: Either release branch creation and/or cherry pick process, can be automated, such automation can be added later after subsequent sprints that shows that the CICD pattern is giving the expected results, similar to PR merges into **main** after full approval.

Bellow picture shows the **dynamics** of a bugfix on a planned release.


1. Planned release, bug found during QA
1.0.0
4. Release Candidate after fully tested
As the bugfix was done from **main** there is no regression risks
3. Cherry Pick Merge (single commit)
Release 1.0.1
main
2. bugfix branch from **main** and merged via **PR**
Development continued

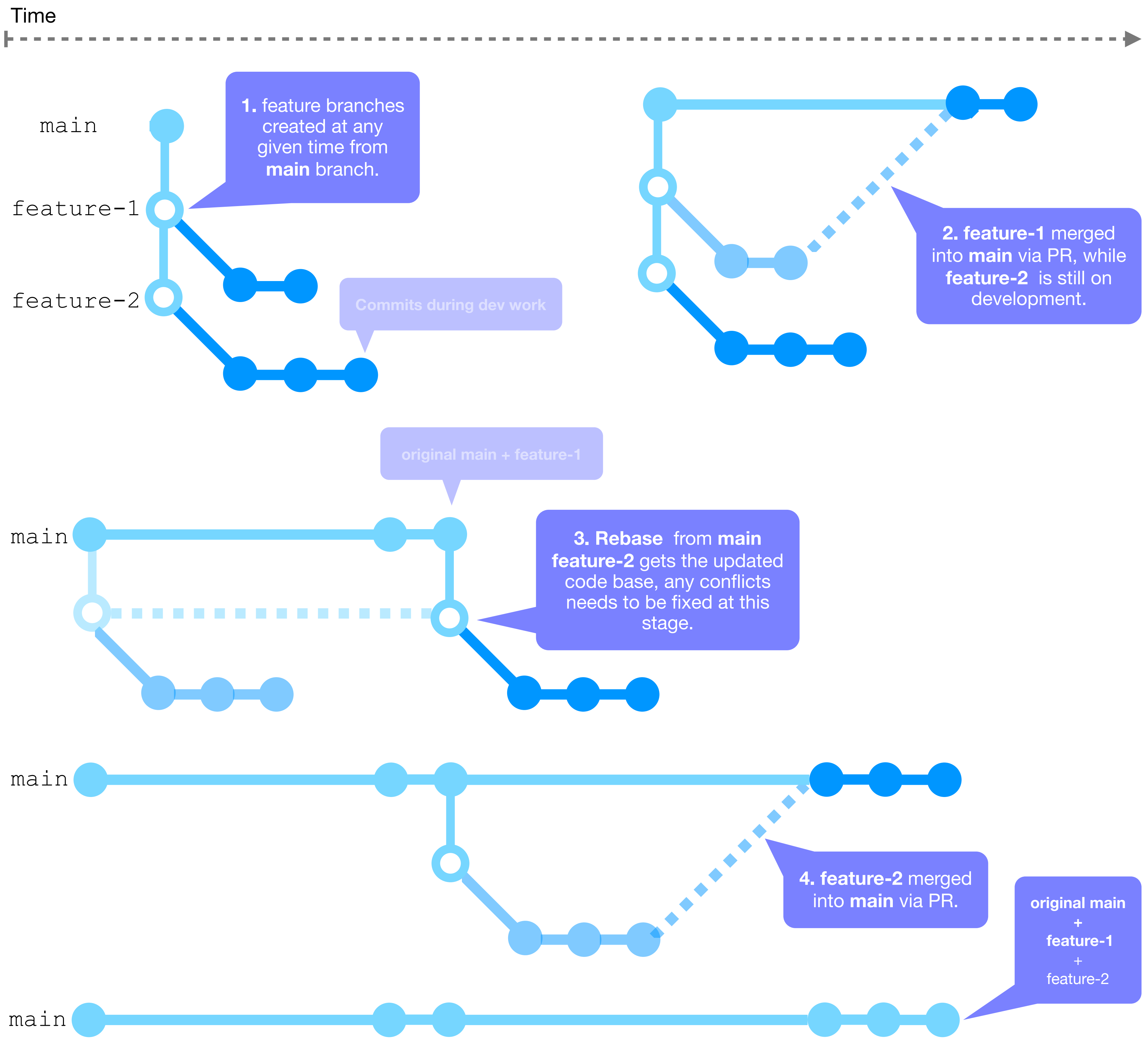
Quote from Google “*So, a release branch is typically a snapshot from trunk with an optional number of cherry picks that are developed on trunk and then pulled into the branch”.*

One of the principles of Trunk Based Development is **No commit pushed to the trunk should jeopardise the ability to go live**.

Iterations needs to happen before gaining reliability and some degree of security, which is based on **code coverage**, plus all the different set of **tests** related with the application itself as the infrastructure, to release them to production on short cycles.

Git Branching Strategies

Git Rebase



Time
main
feature-1
feature-2
1. feature branches created at any given time from main branch.
Commits during dev work
2. feature-1 merged into main via PR, while feature-2 is still on development.
original main + feature-1
main
3. Rebase from main feature-2 gets the updated code base, any conflicts needs to be fixed at this stage.
main
4. feature-2 merged into main via PR.
original main + feature-1 + feature-2
main

On top of the release strategy, the **git squash** can be used, for reference, check *Git Rebase Standard vs Git Rebase Interactive* in git rebase, then, when merging, each PR will have single commit to add into **main**. Depending on the Git provider used, such strategies can be enforced to ensure that the pattern is been followed, see GitHub - configuring pull request merges.

By following the above mentioned branching strategy along with trunk-based-development approach, then the triggering of the deployment pipelines is next.

Continuous Delivery

Continuous Delivery (CD) is the practice of expanding Continuous Integration (CI) to automatically re-deploy a proven build to a **Test** environment for a potential **Production Release**, such test environment can also imply **Quality Insurance (QA)**, as well as **User Acceptance Testing (UAT)**, each one could be an stage of the testing pipeline in the same infrastructure environment.

One of the purposes of CD, is to deliver fast but with a high level of confidence on the Release, such confidence is based on the recreation of isolated environments, similar in features but usually different in capacity, automation could be partial on the later stages of the test pipeline and it depends on the **quantity** and **quality** of the code **coverage** as well as a dynamic set of test scenarios for the infrastructure and the application, as a whole and as granular as possible.

On a three environments strategy, **Dev**, **Test** and **Production** and by following **Trunk Based Development**, usually two pipelines are required to certify a Release, **Dev** and **Test** pipelines, then once the testing is done, an approval process is the one to determine the final **Production** deployment.

Proposed Pipelines Flow:

- **Dev Pipeline:** This pipeline should be triggered when a **Pull-Request** action is **CREATE or UPDATE**, this means, when a Developer has finished its *feature-branch* work (Code, Test, Documentation, etc.) and creates a Pull-Request or updates it, then this pipeline is triggered to proof at least the following:
  - **Build:** The commits added in the feature branch is not breaking the build.
  - **Test**: Any static type of tests to be run before build and after build to comply on the agreed coverage.

- **Deploy:** Deploying into **Development Environment** works, this imply not using development environment for further testing as it will get updated on a higher rate during the Sprint.

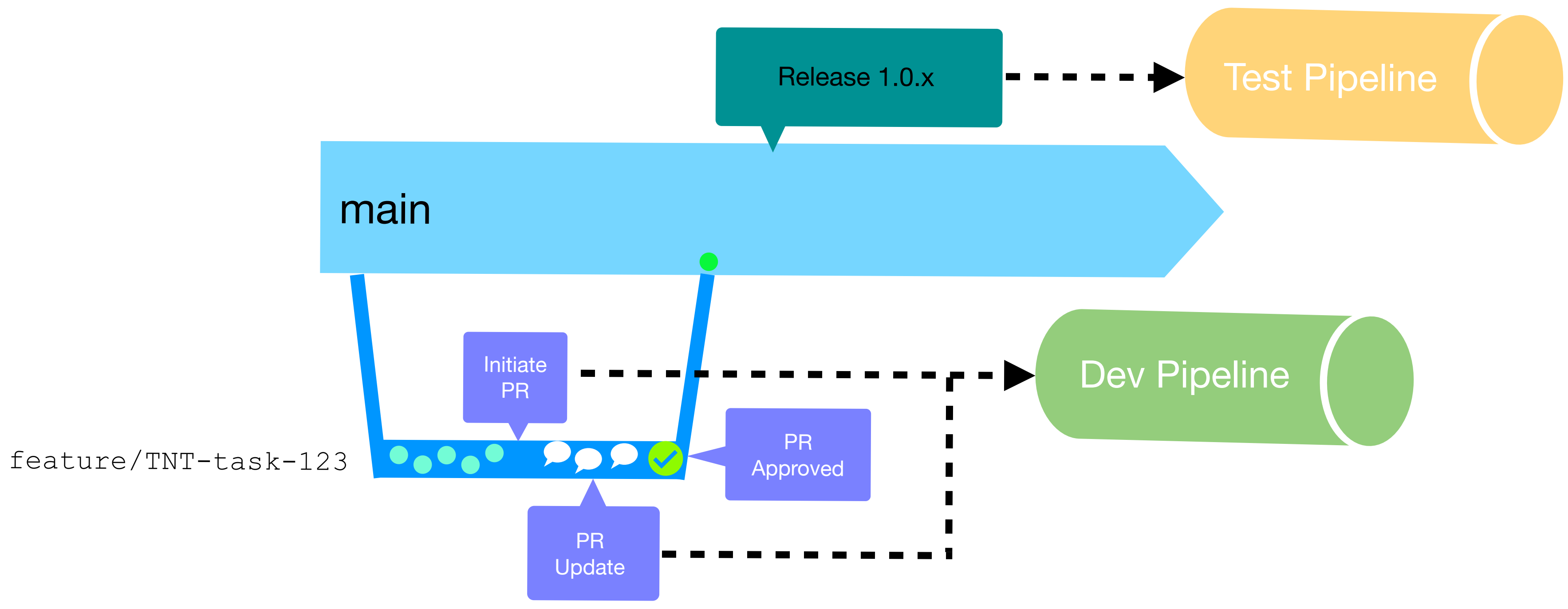
This pipeline must be fully automated at this stage, and a PR will be merged into **main** when:

- The **automation** finished **successfully** plus any other checks set in the Git providers are ok.
- The minimum set peers approved via **Peer-Review** process.

If any of these two conditions needs to be checked again, then the **UPDATE** action will trigger the pipeline cycle.
Once PR approved then whether **manually** or **automated**, such feature, bugfix branch will get merged into **main**, that merge **won't** trigger any pipelines at this stage, as a good practice is to delete such short-lived branches.

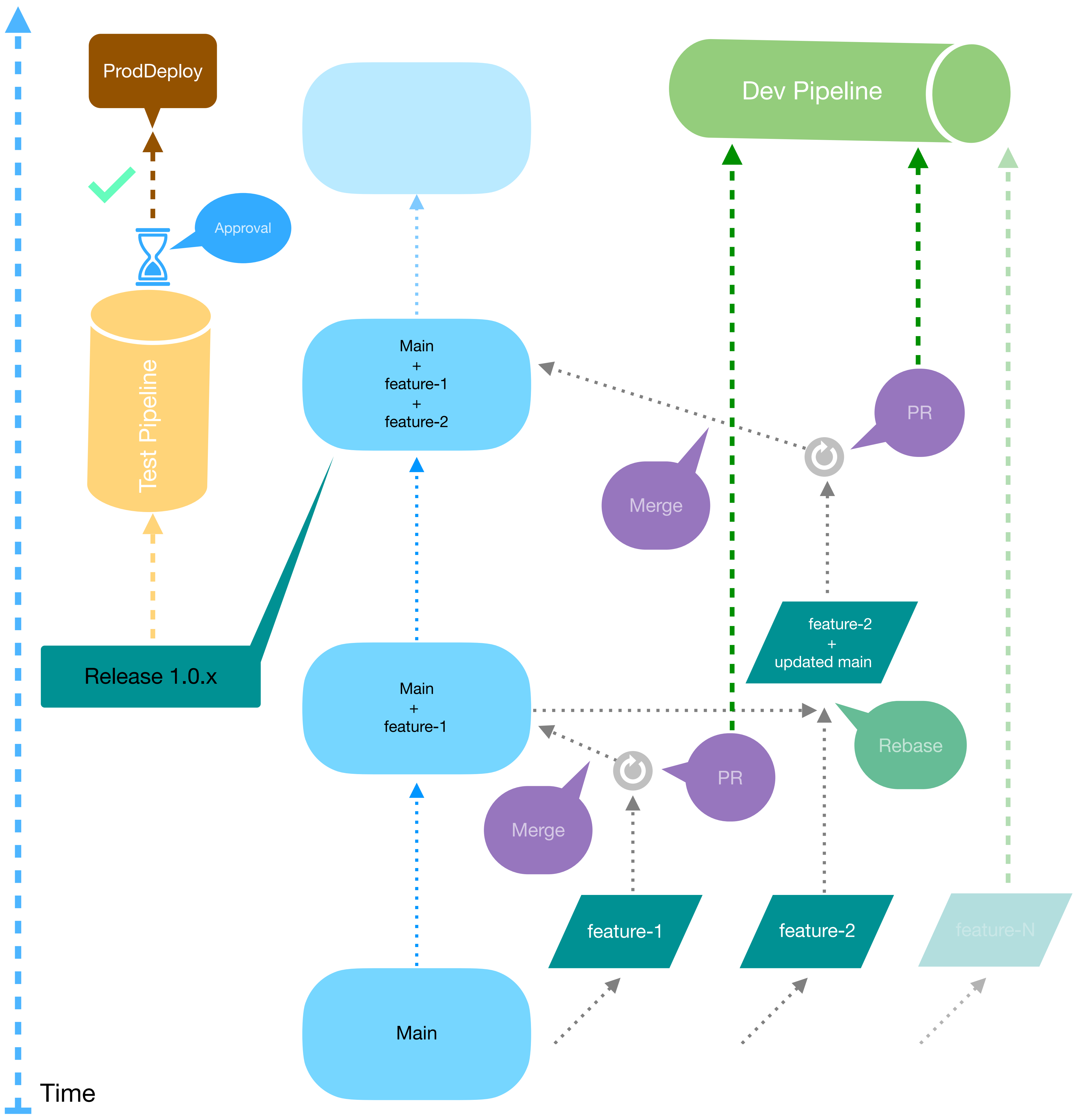
- **Test Pipeline:** This pipeline gets triggered when the **release** branch (release-<version>) is created from **main**, this pipeline will run the same stages as in the development pipeline to proof that at that point in time the code base can be, **built**, **tested** (statically) and **deployed**. Then any set of automated, functional, integration, QA, UAT, tests are executed, usually imply human intervention, to certify the release candidate or to raise bug-fixes.

The creation of the release branch will depend on the **Release Cycle** agreed for each sprint as mentioned in the **CI** section, the main intention at this stage is to certify the release candidate as quick as possible or raise the needed fixes on a context in where the development that is happening is still fresh and aligned.



Example of triggering the pipelines, from PR **create** and **update**, and from Release create.

End-To-End flow at scale:



Different strategies can be followed on the Dev deployment, for full server-less, an entire infra can be deployed on each PR, by following a **suffix**, **prefix** set of strings to identify such infra with the related branch, this allow more experimental approaches when adding new features to existing projects without compromising too much the existent development environment.

Conclusion:

Trunk-Based-Development alongside with **development** and **test** pipelines following the guideline on **Pull-Request** and **Release** triggers, shows a robust and reliable pattern that is scalable from small development teams in where **DevOps**, **Test** and **Development** are all part of the **Software Development Cycle**, or for enterprise teams in where in a single repository different parties add to the codebase, whether for, Infrastructure As Code, Application Code, Testing Code, Documentation, etc. Then each *feature* or *bugfix* is worked within the full context of what the application means. As at the end a CICD Pattern besides the automation and speed of Delivery, is a proven running reusable concept that ensures that such Delivery has high confidence to be deployed in a **Production** environment.

References:

- Trunk Based Development
- Martin Fowler, Continuous Integration
- Martin Fowler, Continuous Delivery
- Git Advanced Tips (concepts of Rebasing vs Merging and Cherry Picking).
- Sam Newman, Building Microservices 2nd Edition, (Chapter 7, build).